by
Ray Duncan

Power Programing

An Introduction to The DOS Protected Mode Interface

I'll never forget how startled I was when I encountered the DOS Protected Mode Interface in its primordial form for the first time. I was sitting in a Microsoft OS/2 2.0 Independent Software Vendor (ISV) seminar in the fall of 1989, with my mind only half-engaged during a talk on OS/2 2.0's Multiple Virtual DOS Machines (MVDMs), when the speaker mentioned in passing that OS/2 2.0 would support a new interface for the execution of DOS extender applications. This casual remark focused my mind remarkably, because at the time I was up to my ears in the production of the Addison-Wesley book *Extending DOS*. The last thing I wanted to hear was that there was an important new protected-mode DOS programming interface that our as-yet-unpublished book would totally neglect.

I asked the speaker for more information, explaining that his mystery interface was about to have a severe impact on a book project near and dear to my heart. In a couple of hours, the Microsoftie returned with a thick document entitled "DOS Protected Mode Interface Specification, Revision Pre-release 0.04"—still warm from the Xerox machine and generously garnished with "CONFIDENTIAL" warning messages. I suspect I made a most amusing spectacle, as I flipped through the pages with my eyes popping and my jaw dropping to the floor. The document I had been handed was nothing more or less than the functional specification of a protected-mode version of DOS!

In retrospect, the fact that Microsoft was cooking up something like the DPMI should have been obvious. Every computer journalist in America, as well as thousands of beta testers, was well aware that the as-yet-unannounced *Microsoft Windows* 3.0 was somehow able to take advantage of extended memory by executing applications in protected mode, even though it ran on top of DOS and used the DOS file system. It was even fairly widely known that you could use a utility program to "mark" the .EXE file headers of old (that is, *Windows* 2.03) applications, and that many such programs would then run (at least for a while) in protected mode, even though they hadn't been recompiled or relinked. But I never saw a word of speculation in print on how this was done, and I never gave it a second thought; I was much too preoccupied with the shifting sands of OS/2 to spend any time puzzling over the internals of a forthcoming version of *Windows*.

■ **The DPMI had its origins in the *Windows* 3.0 development project, but the industry standard DPMI as we know it today only vaguely resembles Microsoft's original internal specification.**

How does *Windows* 3.0 work its magic? The answer is simple: *Windows* 3.0 has a built-in DOS extender! When an application is running in protected mode, *Windows* traps the application's requests for DOS and ROM BIOS services by intercepting its execution of software interrupts (INT instructions). *Windows* then performs any virtual-to-physical address conversions that may be required, copies data from extended memory into conventional memory if necessary, switches the CPU into real mode, and re-issues the function call. When control returns from DOS or the ROM BIOS, *Windows* switches the CPU back into protected mode, converts addresses from physical to virtual and copies data from conventional memory to extended memory as appropriate, and finally hands the machine back to the application.

The process I've just described is perfectly standard DOS extender technology (see my columns in the September 26, October 17, and October 31, 1989, issues of *PC Magazine*), except that these facilities are available to every program running under *Windows* 3.0 on an 80286, 80386, or 80486 CPU.

The implications are immense. It would be a trivial exercise for Microsoft to subsume the *Windows* 3.0 DOS extender into DOS itself, thereby creating an operating system with identical APIs in both real and protected mode. The market for third-party DOS extenders would vanish, the tortuous pathway to protected mode for MS-DOS ISVs would turn into a freeway, the evolution of DOS into a 32-bit, 386-based operating system would become inevitable, and DOS's lifespan would be prolonged nearly indefinitely.

I find it ironic that Microsoft, while it preached the advantages of OS/2's totally new API and bad-mouthed DOS extenders for the last three years, was actually preparing to legitimize DOS extender technology on a massive scale.

THE POLITICS OF PROTECTED MODE

Although the DPMI had its origins in the *Windows* 3.0 development project, and *Windows* 3.0 is the only commercially available DPMI server as this column is written, the industry DPMI standard as we know it today only vaguely resembles that original internal Microsoft specifica-

tion I saw so long ago. Microsoft originally defined the DPMI in two layers: a set of low-level functions for interrupt management, mode switching, and extended memory management, and a higher-level interface that provided access to MS-DOS, ROM BIOS, and mouse driver functionality via protected-mode execution of INT 21h, INT 10h, INT 33h, and so on. The higher-level DPMI functions were implemented in terms of the low-level DPMI functions and the extant real-mode DOS and ROM BIOS interface.

When details of Microsoft's DPMI began to leak out to the general community of MS-DOS developers, the rumors provoked more than a few hard feelings and harsh words, for two very good reasons: First, the vendors of other DOS extenders suspected that Microsoft, having realized that OS/2 wasn't going to replace DOS any time soon, had decided to barge into the market niche they had established so painfully and elbow them out through the sheer weight of its development resources and marketing power. Second, Microsoft had designed the DPMI with total disregard for compatibility with the existing industry standard for DOS-based protected-mode software—the Virtual Control Program Interface (VCPI).

VCPI was developed in 1987 by Phar Lap Software and Quarterdeck Office Systems with a fairly narrow objective: to allow 80386 protected-mode DOS extender applications to coexist with 80386-specific memory managers and expanded memory (EMS) emulators such as Qualitas's *386-to-the-Max* and Quarterdeck Office Systems's *QEMM-386*. In spite of (or because of) its limited goals, the VCPI was quite successful. It became an accepted industry standard in April 1989, and by 1990 virtually every 80386-specific software product on the market supported or was capable of using the VCPI interface—except for Microsoft's products.

The VCPI assumes a client-server model, where the DOS extender application is the client and the EMS emulator is the server. The client invokes the server via an extension of the Lotus/Intel/Microsoft Expanded Memory Specification (EMS) INT 67h interface to switch between real and protected modes, allocate memory, program the interrupt controller(s), and inspect or set the 80386 debug registers.

THE DPMI SERVICES

Function number	Function name	DPMI 0.9	DPMI 1.0
LDT Management Services			
0000h	Allocate LDT Descriptor	*	*
0001h	Free LDT Descriptor	*	*
0002h	Map Real Mode Segment to Descriptor	*	*
0003h	Get Next Selector Increment Value	*	*
0004h	Reserved		
0005h	Reserved		
0006h	Get Segment Base Address	*	*
0007h	Set Segment Base Address	*	*
0008h	Set Segment Limit	*	*
0009h	Set Descriptor Access Rights	*	*
000Ah	Create Code Segment Alias Descriptor	*	*
000Bh	Get Descriptor	*	*
000Ch	Set Descriptor	*	*
000Dh	Allocate Specific LDT Descriptor	*	*
000Eh	Get Multiple Descriptors		*
000Fh	Set Multiple Descriptors		*
DOS Memory Management Services			
0100h	Allocate DOS Memory Block	*	*
0101h	Free DOS Memory Block	*	*
0102h	Resize DOS Memory Block	*	*
Extended Memory Management Services			
0500h	Get Free Memory Information	*	*
0501h	Allocate Memory Block	*	*
0502h	Free Memory Block	*	*
0503h	Resize Memory Block	*	*
0504h	Allocate Linear Memory Block		*
0505h	Resize Linear Memory Block		*
0506h	Get Page Attributes		*
0507h	Modify Page Attributes		*
0508h	Map Device in Memory Block		*
0509h	Map Conventional Memory in Memory Block		*
050Ah	Get Size of Memory Block		*
0800h	Physical Address Mapping	*	*
0801h	Free Physical Address Mapping		*
0D00h	Allocate Shared Memory		*
0D01h	Free Shared Memory		*
0D02h	Serialize on Shared Memory		*
0D03h	Free Serialization on Shared Memory		*
Page Management Services			
0600h	Lock Linear Region	*	*
0601h	Unlock Linear Region	*	*
0602h	Mark Real Mode Region as Pageable	*	*
0603h	Relock Real Mode Region	*	*
0604h	Get Page Size	*	*

CONTINUES

Figure 1: These are the function numbers and names of the DPMI services. The grouping shown here is my own; it is not the same as the grouping found in the DPMI Specification.

If a DOS extender application is loaded and no VCPI server is present, the DOS extender assumes total control of the machine and carries out these hardware-dependent manipulations directly. A more detailed description of the VCPI can be found in Bob Moote's excellent chapter in *Extending DOS* (Addison-Wesley, Reading, Massachusetts, 1990).

In the context of the problems it was designed to solve, the VCPI works extremely well, but it is an inadequate platform for the *multitasking* of DOS extender applications. The VPCI allows client programs to run at the highest privilege level (Ring 0). This makes it impossible for a VCPI server to enforce *device virtualization* (for example, running graphic applications in a window), provide centralized virtual memory management services, or shield one DOS exte[illegible] application from interference from another. Another, somewhat less important drawback of the VCPI is that it is funda-

mentally based on the concept of 80386 hardware paging and therefore can't be implemented on 80286 machines.

The creators of the VCPI were well aware of its limitations and were already hard at work on a second generation specification called *Extended VCPI* (XVCPI), when Microsoft barged onto the scene with the beta-test versions of *Windows* 3.0 and its DPMI. For a few months, it appeared that the fledgling DOS extender market would fragment into two mutually exclusive directions, resulting in additional headaches for software developers, hassles for end users, and juicy fees for lawyers. Luckily, cooler heads prevailed. Microsoft turned control of the DPMI specification over to an industry committee with open membership, and the backers of the XVCPI effort decided to join forces behind the DPMI. Intel, with its understandable enthusiasm for anything that might sell more 80386 chips, was instrumental in bringing about this reconciliation and also took on the responsibility of publishing and distributing the DPMI Specification.

As part of this accommodation, Microsoft agreed to delete the portions of the DPMI that crossed into DOS extender territory—specifically, direct support of the DOS and ROM BIOS interrupts in protected mode. Consequently, DPMI, Version 0.9, the first public version, released by the DPMI Committee in May 1990, defines only the low-level or building-block functions I mentioned earlier. The additional functions in DPMI, Version 1.0, announced in November 1990, are also relatively low-level and are aimed at specific issues of 386-specific memory management and memory sharing (see Figure 1). Naturally, the higher-level or DOS extender interface of *Windows* 3.0 still exists, but it has receded into the twilight zone of undocumented functionality. Undocumented, but hardly unusable—as we shall see.

Although the circumstances of DPMI's birth were somewhat stormy, I think it's fair to say that everyone involved now agrees that the DPMI is a sizable improvement over the VCPI. In many respects, a DPMI server program is similar to a VCPI server, in that it provides mode switching and extended memory man-

THE DPMI SERVICES

PC MAGAZINE

Function number	Function name	DPMI 0.9	DPMI 1.0
Page Management Services (continued)			
0700h	Reserved		
0701h	Reserved		
0702h	Mark Page as Demand Paging Candidate	*	*
0703h	Discard Page Contents	*	*
Interrupt Management Services			
0200h	Get Real Mode Interrupt Vector	*	*
0201h	Set Real Mode Interrupt Vector	*	*
0202h	Get Processor Exception Handler Vector	*	*
0203h	Set Processor Exception Handler Vector	*	*
0204h	Get Protected Mode Interrupt Vector	*	*
0205h	Set Protected Mode Interrupt Vector	*	*
0210h	Get Extended Processor Exception Handler Vector in Protected Mode		*
0211h	Get Extended Processor Exception Handler Vector in Real Mode		*
0212h	Set Extended Processor Exception Handler Vector in Protected Mode		*
0213h	Set Extended Processor Exception Handler Vector in Real Mode		*
0900h	Get and Disable Virtual Interrupt State	*	*
0901h	Get and Enable Virtual Interrupt State	*	*
0902h	Get Virtual Interrupt State	*	*
Translation Services			
0300h	Simulate Real Mode Interrupt	*	*
0301h	Call Real Mode Procedure with Far Return Frame	*	*
0302h	Call Real Mode Procedure with Interrupt Return Frame	*	*
0303h	Allocate Real Mode Call-Back Address	*	*
0304h	Free Real Mode Call-Back Address	*	*
0305h	Get State Save Addresses	*	*
0306h	Get Raw CPU Mode Switch Addresses	*	*
Miscellaneous Services			
0400h	Get DPMI Version	*	*
0401h	Get DPMI Capabilities		*
0A00h	Get Vendor-Specific API Entry Point	*	*
0B00h	Set Debug Watchpoint	*	*
0B01h	Clear Debug Watchpoint	*	*
0B02h	Get State of Debug Watchpoint	*	*
0B03h	Reset Debug Watchpoint	*	*
0C00h	Install Resident Handler Initialization Call-Back		*
0C01h	Terminate and Stay Resident		*
0E00h	Get Coprocessor Status		*
0E01h	Set Emulation		*

ENDS

unlike a VCPI server, a DPMI server runs (or can run) at a higher privilege level than its clients—using the hardware to enforce a "kernel/user" protection model. This allows a DPMI server to support demand-paged virtual memory and maintain full control over client programs' address spaces and access to the hardware. Furthermore, the DPMI's functions for memory and interrupt management are far more general than those in the

mented on 80286 machines.

Over the long run, existence of the DPMI will simplify life enormously for vendors of DOS extenders, multitasking control programs, EMS emulators, and protected-mode programming tools. During the transition, however, the developers of DOS extenders in particular are going to find their life considerably more complicated. Not only will each DOS extender have to be capable of performing all

tions on its own, but it will have to be able to coexist peacefully with Microsoft's XMS driver (HIMEM.SYS), VCPI servers such as *QEMM* 5.0, and DPMI servers such as *Windows* 3.0—a goal similar to writing a single graphic application that could execute equally well under Digital Research's *GEM*, *Microsoft Windows*, OS/2's Presentation Manager, or even without an operating system!

THE DPMI INTERFACE

The DOS Protected Mode Interface (DPMI) Specification can be obtained at no charge from Intel Literature Sales, P.O. Box 58130, Santa Clara, CA 95052, or telephone 800-548-4725. The function calls supported by the DPMI fall into seven general categories, as shown in Figure 1:

- The *LDT Management* functions allow a program to manipulate its Local Descriptor Table (LDT). A program can allocate and free descriptors (and their associated selectors), inspect or modify the descriptors, map real-mode addresses onto protected-mode selectors, change the access rights of a segment (for example, from read-write to read-only), and obtain a read/write data selector (alias) for an executable selector.
- The *DOS Memory Management* functions provide a protected-mode interface to the real-mode MS-DOS INT 21h Functions 48h (Allocate Memory Block), 49h (Free Memory Block), and 4Ah (Resize Memory Block). Using these functions, a protected-mode program can obtain memory below the 640K limit that it can use to exchange data with MS-DOS itself, TSRs, ROM BIOS device drivers, and other real-mode programs.
- The *Extended Memory Management* functions are used to allocate, resize, and release blocks of physical memory above the 1MB boundary. The functions are low-level in that they do not allocate selectors or build descriptors for the extended memory blocks; the program must allocate selectors and map the memory onto the selectors with additional DPMI calls. On an 80386 or 80486 with paging enabled, the allocated blocks are always a multiple of 4K. Named shared memory blocks with serialized access are also supported.
- The *Page Management* functions allow memory to be locked or unlocked for swapping on a page-by-page basis in terms of the memory's linear address. There are also functions to query the page size supported by the host CPU and to tune the system's behavior by marking pages for discard or immediate replacement. These functions are only relevant when the host machine is an 80386 or 80486.
- The *Interrupt Management* functions support interception of software or hardware interrupts that occur in real mode or protected mode, installation of handlers for processor exceptions and faults (such as divide by zero and overflow), and maintenance of a separate "virtual interrupt flag" for each active process.
- The *Translation Services* provide a mechanism for cross-mode procedure calling. They allow a protected-mode program to transfer control to a real-mode routine by either a simulated far call or a simulated interrupt, and pass parame[ters] by value or by reference. A protected-mode program can also declare an entry point (known as a Real Mode Call-Back)

HELLOPMW.C — **COMPLETE LISTING**

```
/*
    HELLOPMW.C  ---  "Hello Protected Mode World!"
    Illustrates use of the DPMI interface to switch into
    protected mode, and use of Windows 3's built-in DOS
    Extender to print a message in protected mode.

    Compile in SMALL MODEL with:  CL HELLOPMW.C

    Execute under Windows 3.0 only!
*/

#include <stdio.h>

unsigned modeswitch();

main()
{
    unsigned saveCS, saveDS;

    _asm mov saveCS,cs              ; store real mode CS
    _asm mov saveDS,ds              ; and DS for display

    printf("\nHello, real mode world! \tCS=%04xh DS=%04xh",
        saveCS, saveDS);

    if(modeswitch())                // attempt mode switch
    {
        puts("\nDPMI not available or memory allocation failed.");
        exit(1);
    }

    _asm mov saveCS,cs              ; store protected mode CS
    _asm mov saveDS,ds              ; and DS for display

    printf("\nHello, protected mode world! \tCS=%04xh DS=%04xh\n",
        saveCS, saveDS);

    _asm mov ah,4ch                 ; exit directly to DOS to avoid
    _asm int 21h                    ; GP fault in RTL cleanup code
}

/*
    Call DPMI to switch from real mode to protected mode.
    Returns false if mode switch successful, true if no DPMI
    present or if memory allocation for DPMI data area failed.
*/
unsigned modeswitch(void)
{
    void far *switchpoint;                  // far pointer to DPMI
                                            // mode switch entry point
    _asm
    {
        mov     ax,1687h                    ; get DPMI mode switch
        int     2fh                         ; entry point...

        or      ax,ax                       ; bail out if no DPMI
        jnz     exitlabel
        mov     word ptr switchpoint,di     ; save entry point
        mov     word ptr switchpoint+2,es
        mov     bx,si                       ; allocate DPMI private
        mov     ah,48h                      ; data area
        int     21h
        jc      exitlabel                   ; jump, allocation failed
        mov     es,ax                       ; pass segment of data area
        mov     ax,0                        ; indicate we are 16-bit app
        call    switchpoint                 ; switch to protected mode
        mov     ax,0                        ; signal success
    }

    exitlabel: ;                            // any "no return value" warning
}                                           // message here can be ignored
```

Figure 2: Here's the source listing for HELLOPMW.C, a Microsoft C program that uses the DPMI to switch into protected mode and then uses the built-in *Windows* 3.0 DOS extender to display text in protected mode.

which can be invoked by a real-mode program with an implied mode switch.

- The *Miscellaneous Services* include address conversions, coprocessor management, debugging support, and a function to get the DPMI version number.

Nearly all of these DPMI functions are intended only for DOS extenders; they would ordinarily never be called directly by an application program. But there are two DPMI functions that we can put to immediate good use: one that indicates whether a DPMI server is present, and one that switches from real to protected mode. These functions allow us to write programs (using ordinary DOS programming tools) that run in protected mode on top of *Windows* 3.0's built-in DOS extender.

First, we need to check for the existence of the DPMI host by executing an INT 2Fh with the value 1687h in AX. If no DPMI server is present, AX is returned unchanged or with some nonzero value. If a DPMI server is available, the following values are returned:

```
AX    = 0
BX    = Flags (bit 0 = 1 if
         32-bit programs supported)
CL    = Processor type
         (02H=80286, 03H=80386,
          04H=80486, 05H=80586)
DH    = DPMI major version number
DL    = DPMI minor version number
SI    = number of paragraphs
          required for DOS Extender
          private data (may be 0)
ES:DI = DPMI mode switch entry
          point
```

Next, we call the entry point whose address was returned in ES:DI to switch the CPU from real mode or Virtual 86 mode into protected mode. Bit 0 of AX is 0 if our program is a 16-bit application, and it's 1 if the application is 32-bit; the remaining bits in AX are reserved. ES must point to the base of a scratch area as large as the size returned in SI from the DPMI existence call. Upon return, the CPU is in protected mode; CS, DS, and SS have been loaded with valid selectors for 64K segments that map to the same physical memory as the original real-mode values in CS, DS, and SS; ES contains a selector that points to the program's PSP with a 256-byte segment limit; FS and GS contain zero (the null selector) if the host machine is an 80386 or 80486; and all other registers are preserved.

Figure 2 contains a simple C program that illustrates the use of these two calls, as well as *Windows* 3.0's support for INT 21h calls in protected mode. You can use the ordinary Microsoft C Compiler and Linker to build this program, though you must be sure to compile it in the Small Memory Model. You need *Windows* 3.0 running on an 80286, 80386, or 80486 to run the program. My thanks to Andrew Schulman for pointing out that such programs are possible! In the next issue, we'll discuss the DPMI and the *Windows* 3.0 DOS extender in more detail.

THE IN-BOX

Please send your questions, comments, and suggestions to me at any of the following e-mail addresses:
PC MagNet: 72241,52
MCI Mail: rduncan
BIX: rduncan ■